

Constraint satisfaction algorithms: edition of timetables in the license-master-doctorate system

Maurice Comlan¹, Corentin Allohoumbo²

¹Department of Business Computing, National School of Applied Economics and Management, University of Abomey-Calavi, Cotonou, Bénin

²Department of Computer Engineering and Telecommunications, Polytechnic School of Abomey-Calavi, Cotonou, Bénin

Article Info

Article history:

Received Apr 6, 2023

Revised May 13, 2023

Accepted May 28, 2023

Keywords:

Constraint satisfaction problem
Genetic algorithm
License-master-doctorate system
Scheduling
Simulated annealing

ABSTRACT

In this paper, we studied some algorithms for solving constraint satisfaction problem (CSP) and then applied them to solve the problem of generating schedules in a university setting. In other words, we studied the genetic algorithm, the simulated annealing, the hill climbing, a hybridization of the genetic algorithm and the simulated annealing as well as a hybridization of the genetic algorithm and the hill climbing. These algorithms have been tested on the problem of scheduling in a university environment. The hybrid uses hill climbing or simulated annealing to improve each individual in the starting population to a certain stopping point. These individuals are then sent to the genetic algorithm. Our results show that the hybridization of the genetic algorithm with a metaheuristic gives better execution time and performs better as the problem size increases compared to the classical genetic algorithm.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Maurice Comlan

Department of Business Computing, National School of Applied Economics and Management

University of Abomey-Calavi

03 BP 1079 Cotonou, Bénin

Email:maurice.comlan@uac.bj

1. INTRODUCTION

Constraint programming (CP) is a programming paradigm that appeared in the 1970s and 1980s and allows solving large combinatorial problems such as planning and scheduling problems. In constraint programming, the modeling part is separated using constraint satisfaction problems (CSP), and the resolution part relies on the active use of the problem's constraints to reduce the size of the search space (referred to as constraint propagation) [1], [2]. There are several problems that can be solved using constraint programming, including university timetable construction. Universities have always faced the challenge of providing schedules for teachers and students. A naive solution would be to let each teacher choose the schedule that suits them without considering other courses. While this may make it easier for the teacher, it creates many collisions in student schedules [3]. The approach is to try to avoid these collisions while trying to satisfy everyone's needs. Solving this problem manually takes a lot of time, depending on the size of the university. To reduce this workload, researchers have tried to make the method easier by automating it using various types of algorithms and techniques. The problem is called the university course timetabling problem (UCTP) [4], [5]. UCTP is an NP-hard combinatorial problem [6], [7], partly because there are many variables involved such as teachers and students with constraints between them, and partly because each university has its own variables and constraints. The UCTP involves assigning course events, students, and teachers to a time slot and classroom while violating the

fewest constraints possible [8].

The course scheduling problem is defined as the problem of placing resources into limited time slots and locations that are compatible with certain constraints. The scheduling of teaching is a problem that involves creating a schedule for groups of students, rooms, and teachers to meet certain constraints. School scheduling includes exam timetabling and university course scheduling. Creating course schedules in universities nowadays is a real problem as the number of students keeps increasing, and some courses impose their constraints. This task is mostly done by humans and becomes tedious as the number of parameters to consider exceeds a certain threshold. We mostly witness some problems, such as a teacher having two courses at the same time with two different classes or majors on the same day. Therefore, it would be wise to have a system that helps us construct course timetables in universities while respecting the hard constraints. Course scheduling is considered one of the main concerns of university staff before the start of the semester or academic year. Manual scheduling is tedious and usually takes several weeks to create an appropriate schedule. The biggest difficulty is related to the problem's size. The algorithm must take into account several conflicting criteria and conditions, often for an enormous number of classes, teachers, and courses. Moreover, the structure of the timetable and quality criteria differ from one country to another, or even within the same school. For these reasons, the problem of university timetabling has been the subject of many studies within the research community. With the implementation of the license-master-doctorate (LMD) reform in the education system, universities face difficulties in creating suitable course timetables for all stakeholders. Even if they succeed, they face a lot of problems, such as hard constraints being broken. In the field of education with the LMD reform [9], universities are struggling to create a suitable course schedule for all stakeholders. Even if they manage to do so, they are confronted with many problems such as hard constraints that are broken. At a time when all universities using the LMD system are facing this problem and would like to have a system that helps them, it would be wise to study this problem in order to propose an acceptable system capable of offering a viable solution using constraint satisfaction algorithms. The objective of this work is to study constraint satisfaction algorithms in order to apply them to build a system that generates schedules while respecting the hard constraints that are defined.

The aim of this paper is to show that the hybridization of the genetic algorithm with a metaheuristic gives better execution time and performs better as the problem size increases compared to the classical genetic algorithm. In the remainder of this paper, the section 2 discusses about constraint programming in general before discussing about the genetic algorithm, simulated annealing, hill climbing and the hybrid genetic algorithm. The section 3 talk about our approach and presents our system architecture, the section 4 presents the obtained results and in the final section (section 5) of the paper we find its conclusion.

2. RELATED WORKS

2.1. Fitness function

The quality of a solution in this context is determined by a fitness function, which plays a crucial role in evaluating the effectiveness of the algorithms employed. This fitness function calculates the number of stress breaks while assigning weighted values to different types of stresses. The resulting value serves as a metric for assessing the overall quality of the solutions generated by these algorithms. A solution is considered workable if it adheres to all the hard constraints without violating any of them. The algorithms aim to maximize the fitness function, where a perfect solution achieves a fitness score of 1. A fitness score lower than 1 indicates that the solution violates one or more constraints, highlighting areas where improvements are needed. To illustrate these constraints using the example of UCTP, some of the constraints include: i) lessons can only be scheduled in available (i.e., unoccupied) rooms; ii) each teacher can instruct only one course at any given time; iii) the classrooms assigned must be spacious enough to accommodate the group of students assigned to the class; and iv) students cannot be expected to attend two different classes simultaneously. In the context of this problem, a "clash" refers to a hard constraint violation, signifying situations where constraints have been decisively breached. This emphasizes the importance of finding solutions that not only optimize the fitness function but also adhere to these critical constraints, ensuring a feasible and effective schedule for university courses.

$$fitness = \frac{1}{1 + \sum clash} \quad (1)$$

2.2. Neighbor function

The following neighbor algorithm is used in the internal repeat loop of the SA. In this neighboring algorithm, we try to play on the time intervals in order to eliminate as much as possible collisions due to time intervals. To do this, we model the time intervals day by day as a two-dimensional matrix. First, the algorithm evaluates the collision using the fitness function but while considering that the violations time interval. For each pair (i, j) that are in collision, we try to move them in another cell while taking into account its current position. The possible actions are: left, right, up, and down. Then the best movement is chosen after checking that the temperature has decreased. The Figure 1 shows us what modeling prices as a function of intervals looks like of time and day. In this example, we have assumed that the university teaches from 7 a.m. to 7 p.m. (monday to saturday).

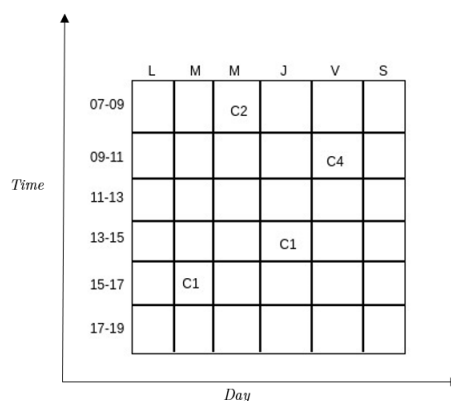


Figure 1. Matrix modeling of courses based on days and hours

2.3. Constraint programming

A constraint is a logical relationship established between different variables, each taking its own value in a set of its own, called a domain. A constraint on a set of variables restricts the values that its variables can take simultaneously. A constraint is declarative and relational since it defines a relationship between variables without specifying operational procedure to ensure this relationship. The arity of a constraint is equal to the cardinality of its set of variables, called its scope. For example, the binary constraint $(x, y) \in (0, 1), (1, 2)$ specifies that the variable x can only take the value 0 if the variable y takes the value 1 and that the variable y can take the value 2 only if x takes the value 1. In optimization problems, there typically exist two categories of constraints: hard constraints and soft constraints. Hard constraints are conditions that are mandatory to fulfill in order to find a viable solution. They often represent non-negotiable rules, such as ensuring no resource overlaps. On the other hand, soft constraints are considered optional restrictions and are not obligatory to meet. Instead, they are employed to evaluate the quality of a solution. It is worth noting that soft constraints can be formulated in a manner that makes it challenging or even impossible to satisfy all of them simultaneously [10]. As an example of the constraints of the UCTP we have: all events are scheduled in a room and a time slot [11]; students and teachers cannot attend two events at the same time [12]; a student does not have a class in the last time slot of a day [12].

In constraint programming, the problems are modeled using decision and constraint variables, where a constraint is a relation between one or more variables that limit the values that each of the linked variables can take simultaneously by coercion. The solution finding algorithms, in CP, are generally based on the propagation of constraints [13], to reduce the number of candidate solutions to be explored, as well as on a systematic search among the different possible assignments of each of the variables. Such algorithms guarantee to find a solution, when it exists, and make it possible to prove that there is no solution to a problem if they have not found a solution by the end of the search exhaustive. In the case of finite domain resolution, it is in theory possible to enumerate all the possibilities and check if they violate the constraints or not, this method is called generate and test [14]. However, this proves impractical for medium-sized problems in because of the large number of possible combinations. One of the main parts of the resolution, called “filtering”, is intended to avoid this exhaustive enumeration. It consists in deducing from the constraints the impossible values. When a variable has only one candidate, it is instantiated (i.e. this value is assigned to it). However, filtering alone does not allow

instantiate all the variables, and it is therefore necessary to split the problem into several parts (for example by instantiating a variable at each of its possible values) and restarting the filtering on one of these parts and this, recursively until obtaining the instantiation of all variables. When the filtering detects that the partial instantiation violates a constraint, we use generally then a rollback mechanism in order to question the last choice carried out. This series of problem breakdowns can be represented as a tree. The goal of the search is to browse this tree (building it as you go) until you find a solution to the problem while the filtering consists in “pruning” this tree by removing all the parts only leading to contradictions [1].

2.4. Genetic algorithm

The genetic algorithm (GA) is an optimization method grounded in the principles of genetics and natural selection. It is frequently employed to discover optimal or nearly optimal solutions to complex problems that would otherwise require an extensive amount of time to solve. GA finds widespread use in tackling optimization problems in fields like research and machine learning. Nature has long served as a profound source of inspiration for humanity, and GA are a prime example of this influence. These algorithms operate as search methods rooted in the concepts of natural selection and genetics, falling under the umbrella of evolutionary computation, a broader computational discipline. In a GA, a population of potential solutions to a given problem is initially established. These solutions then undergo processes resembling recombination and mutation, akin to natural genetic processes, resulting in the creation of new offspring. This cycle repeats across multiple generations. Each individual within the population, or candidate solution, is assigned a fitness value based on its performance in terms of the objective function. The fittest individuals are more likely to mate and produce offspring, mirroring the Darwinian theory of “survival of the fittest” [15].

This approach allows us to continuously refine and “evolve” superior individuals or solutions across generations until we meet a predefined stopping criterion. Genetic algorithms exhibit an inherent level of randomness, yet they outperform random local search techniques, where solutions are randomly tried and the best ones are retained, due to their ability to also leverage historical information [16]. Genetic algorithms, often abbreviated as GAs, serve as potent and versatile optimization tools that emulate the principles of evolution. They possess the capability to discover globally optimal solutions, even within highly intricate search spaces. GAs operate on a population of solutions selected based on their quality, serving as the foundation for generating a new generation of solutions through a combination process called crossover or modification through mutation of the existing individuals. Traditionally, the search mechanism in GAs remains domain-independent, implying that crossover and mutation operators lack knowledge regarding what constitutes a good solution [17].

2.5. Simulated annealing

Simulated annealing (SA) stands as a local search algorithm that has been employed for tackling NP-hard optimization problems since the 1980s. Local search algorithms rely on two key components: a neighborhood function and a fitness function, with the aim of locating a local optimum. SA introduces a temperature control parameter into the mix, allowing it to break free from local optima and seek solutions closer to the global optimum. This concept draws its inspiration from the annealing process observed in materials. In the annealing process of materials, when a substance transitions from a liquid state to a solid one, the constituent particles exhibit random movement. However, in the solid state, these particles align themselves in an ordered fashion with minimal energy. The ideal state is reached only if the material’s temperature is initially high enough, and subsequent cooling occurs at an appropriate rate. Failing to meet these conditions results in the material freezing in a stable but imperfect state. SA operates as a Monte Carlo method and necessitates an initial solution to initiate the optimization process [18]. This initial solution can be generated randomly or using solutions generated by another algorithm [19]. SA is a single-solution, trajectory-based metaheuristic, meaning it has one solution that is modified and improved until the algorithm stops. Algorithms that always seek the best solution are likely to get stuck in a local optimum. The advantage of SA is that the fitness of an accepted neighboring solution may decrease as iterations progress, as well as increase, to avoid being stuck in a local optimum [20]. The SA implementation uses the same data structures to store schedules as GA and a static cooling program by Abdullah *et al.* [19]. The program of cooling is described in (2) where T is the current temperature, T_0 is the temperature initial, μ is a constant and k is number of iterations. A negative μ guarantees that the temperature will decrease over time and the probability of accepting the worst solutions decreases over time as the SA nears the end of the search. The parameters for SA were calculated by testing different values in order to choose an worst-case solution and the algorithm had to perform 200 (k_{max}) iterations.

$$T = T_0 * e^{-*k} \quad (2)$$

2.6. Hill climbing

Hill climbing (HC) is a heuristic search technique commonly utilized in artificial intelligence for addressing mathematical optimization problems. When provided with a substantial set of inputs and an effective heuristic function, HC endeavors to locate a satisfactory solution to the problem, although it may not necessarily identify the global optimum. In this context, mathematical optimization problems typically entail the HC approach being applied to situations where the objective is either to maximize or minimize a given real function by selecting values from the provided inputs. A crucial element in HC is the heuristic function, which plays the role of assessing and ranking all feasible alternatives at each branching step within the search algorithm, based on the available information. This function aids the algorithm in making informed decisions about the most promising path to pursue among the various options. It's worth noting that HC belongs to the category of generate-and-test algorithms, which typically follow a three-step process: i) generate a potential solution, ii) evaluate if it satisfies the desired criteria, and iii) if a satisfactory solution is found, terminate the process; otherwise, return to step 1 and continue the search.

There are several variations of HC, including steepest ascent HC, first-choice HC, and simulated annealing [21]. In steepest ascent HC, the algorithm examines all potential moves from the current solution and selects the one that results in the greatest improvement. In first-choice HC, the algorithm randomly picks a move and accepts it if it leads to improvement, regardless of whether it's the best move. Simulated annealing is a probabilistic variation of HC that occasionally allows the acceptance of worse moves to avoid becoming trapped in local maxima. HC is applicable to a wide range of optimization problems, including scheduling, route planning, and resource allocation. However, it has limitations, such as a tendency to become stuck in local maxima and limited exploration of the search space. Therefore, it is often combined with other optimization techniques like genetic algorithms or simulated annealing to overcome these limitations and improve search results [22].

2.7. Genetic hybrids algorithms

Hybrid algorithms that combine GA and SA have proven effective in solving various problems, not limited to the UCTP. These algorithms leverage the unique strengths and weaknesses of GA and SA: GA is proficient at exploring extensive search spaces and finding reasonable solutions but often falls short of achieving the optimal solution within a reasonable timeframe; SA, conversely, focuses on a single solution and doesn't explore the search space comprehensively, potentially missing valuable solution areas [23]. By capitalizing on the complementary features of these algorithms, hybrid approaches can yield promising results. Talbi and Muntean [21] proposed an intriguing idea: using GA to identify high-performance regions in a vast search space and then employing a local search algorithm to pinpoint the optimal solution. For instance, Elhaddad and Sallabi devised a hybrid GA-SA method to tackle the traveling salesman problem. Their algorithm seamlessly switched between GA and SA when either faced challenges. They employed a small population size and multi-crossover, generating a substantial number of offspring relative to the population size. This hybrid approach yielded impressive outcomes when tested on standard benchmark datasets [23].

Genetic algorithms are iterative optimization procedures that repeatedly apply operators such as selection, crossing, and mutation to a group of solutions until a convergence criterion is met. In a GA, a research point (solution), a parameter in the search space with dimensions (variables), is encoded in a chain, which is called a chromosome in biological systems. The chain or chromosome is made up of characters that are analogous to k genes. A set of several points searches where a set of chromosomes (or individuals) is called a population. Each iterative step where a new population is obtained is called a generation. A hybridized GA with a local search procedure is called a hybrid genetic algorithm.

3. MATERIAL AND METHODS

3.1. Material

The experiments and tests were carried out on a computer with the following characteristics: operating system: Debian 10; architecture: 64 bits; processor: AMD Ryzen 7 Pro 3700 CPU @ 3.6 GHz × 8; RAM memory: 64 GB. To carry out our experiments, we used a dataset that we divided into three sizes. We have the small size, the medium size and the large size. The Table 1, shows us the number of teachers, groups of

students, courses and time intervals according to the size of the problem. In this study, we used data from the Department of Computer and Telecommunications Engineering in addition to the two preparatory years of the Polytechnic School of Abomey-Calavi.

Table 1. Data set used for our experiments

Parameter	Small	Medium	Large
Teachers	10	28	41
Rooms	2	8	8
Courses	16	66	119
Time slots	16	33	36
Student groups	2	4	7

3.2. Modeling and implementing the UCTP

The planning of courses for a university timetable is often encountered with constraints (hard and soft) due to the diversity compared to a schedule where the demands are very limited [24]. Problems related to constraints hardships must be resolved to produce a working solution. To optimize performance In planning, it is important to address the problems associated with flexible constraints. However, a cautious approach should be formulated without compromising solutions to the constraints. Hard to minimize serious system disruption. As such, a simple planning system such as for a small university system cannot be successfully applied to an organization. complex organization without turning to a different approach for quick results and optimal [25]. There may be conflicts between several soft constraints and therefore a compromise will have to be reached between them. For example, a class might have 12 students, and as that such, the soft constraint will allocate a suitable classroom that can accommodate the students. However, there may be a classroom the teacher prefers that can accommodate up to 45 students. The class planner will hopefully find a preferred setup if the teacher's preference is taken into account in the flexible constraints.

For each class programmed by this application the following hard constraints will be taken in consideration: i) classes can only be scheduled in free rooms; ii) a teacher can only teach one course at a time; iii) classrooms should be large enough to accommodate the group of students; and iv) a group of students cannot attend two classes at the same time. To be able to solve the problem, we will have to use the following data: professors available; classrooms available; time intervals; student groups and the subjects taught.

3.3. Algorithms parameters

Various parameters underwent testing to identify suitable configurations for comparing algorithms. These tests were conducted using extensive assembly data, employing both a time constraint and a fitness score as stopping criteria. The fitness score was set at 1, and the time limit was restricted to one hour. The objective was to determine parameter configurations that would allow the algorithms to attain the fitness goal before reaching the time limit. Each combination of parameters was tested nine times to account for the inherent variability associated with the use of probabilistic heuristics. However, due to the sheer number of parameter combinations and the considerable time required to execute each combination along with the tests, it became unfeasible to complete this comprehensive analysis. The parameters that were tested for GA were probability of mutation, size population and selection size, as shown in Table 2. All combinations of parameters were tested. Generally speaking, we have limited the number of generations to 200 for the genetic algorithm. Regarding the SA and the HC we have used the parameters which can be found in the Table 3.

Table 2. Test parameters for GA

Parameter	Small	Medium	Large
Population size	100	200	600
Mutation probability (P_m)	0.1	0.05	0.01
Crossover probability (P_c)	0.7	0.7	0.7
Selection size	1	3	5

Table 3. Test parameters for SA and HC

Parameter	Value
T_0	1
k	200
μ	0.005

4. RESULTS

In order to achieve our objectives, the methodology followed includes several key steps, including results and the various discussions and interpretations that result from them are presented in this chapter. In detail, this chapter will present the results of the parameter test for the GA, HCGA, and SAGA and the comparison between the three algorithms. As described in the method, each parameter setting was executed nine times with a fitness value of 1 and a one hour maximum. The Table 4 shows the average execution time of the different algorithms on each size of problem. The Table 5 shows the average number of generations it took for each algorithm to get a solution while depending on the size of the problem shows us the average number of generations it took for each algorithm before getting a solution all depending on the size of the problem.

The Figure 2 shows the elapsed time based on the size of the problem. The Figure 3 shows the decrease in the number of collisions as a function of the number of generations. The Figure 4 shows the evolution of the average fitness of the population as a function of the number of generations. The stagnant behavior of the GA in the later stages of execution was expected given the findings of Thierens [11]. In testing, all algorithms were able to find a solution within one hour regardless of the size of the problem, but the execution time will vary depending on the size of the problem. The execution time is exponential which explains the time taken by the different algorithms. We can see that SAGA gives better execution time compared to HCGA and GA based on the Table 4. The slowness of the algorithms can be explained by the fact that the GA uses a population and does a lot of work between each generation. Based on the Figure 2 we deduce that the SAGA converges faster than the two algorithms. We also note that the simple GA falls enormously in an optimum room. This is explained by the parts of the curve which are almost a line.

Table 4. Average execution time of algorithms in seconds depending on the size of the problem

Size	GA	SAGA	HCGA
Small	11	11	6
Medium	131	168	175
Large	2209	1713	2642

Table 5. Average number of generations depending on the size of the problem

Size	GA	SAGA	HCGA
Small	5	5	3
Medium	15	20	21
Large	125	97	149

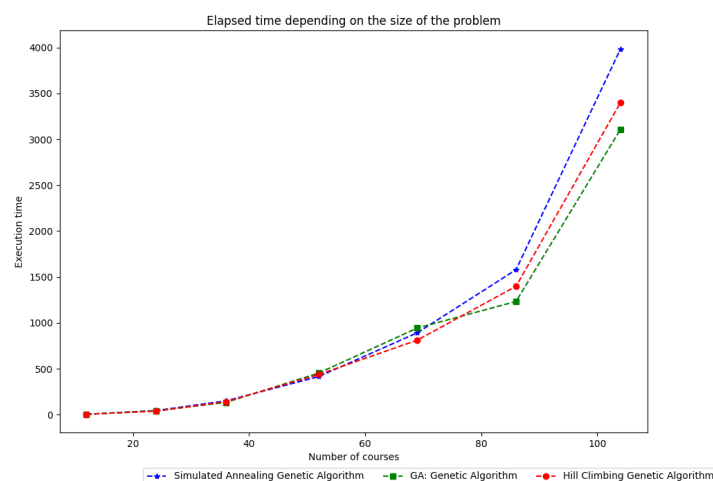


Figure 2. Elapsed time depending on the size of the problem (large size)

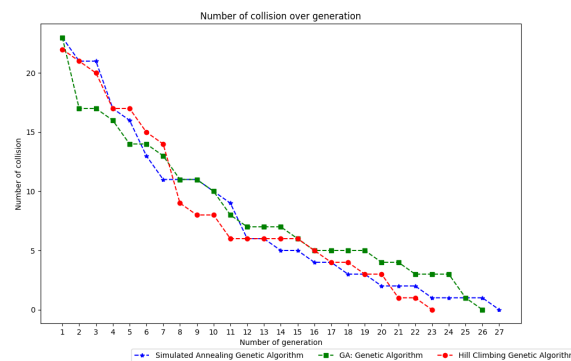


Figure 3. Number of collisions according to the generation (large size)

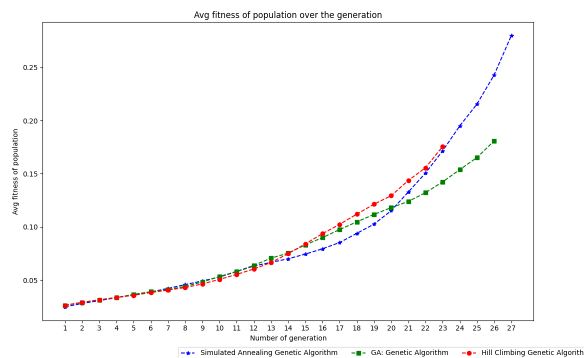


Figure 4. Average fitness of the population according to the generation

5. CONCLUSION

CP is a programming paradigm that allows to solve large combinatorial problems such as that planning and scheduling issues. In this paper, we focused on the study of genetic algorithms. Specifically, we have studied the algorithm classical genetics then two hybrid algorithms that are HCGA and GASA. A hybrid GA is the merger of a metaheuristic or local search with a normal GA. In our study, we opted for high-level hybridization, which involves running each algorithm sequentially. We have applied these algorithms to propose a system of the development of timetables in a university environment. We first modeled and standardized the problem for the different algorithms in order to have the same bases for compare them. Based on our results, the three algorithms manage to find a result at problem within the timeframe we set for our testing. We have the SAGA in particular which has the best results (execution time and total number of generations) for the broad problem. So we can say that genetic algorithms are powerful algorithms able to intervene in the resolution of most satisfaction problems. Based on our results, we can say that the study carried out can be further improved in the sense that we can try to integrate the management of flexible constraints in order to have a objective function which takes into account both hard and soft constraints. In addition, the time performance can be improved by using an alternate way of encoding chromosomes or using backtracking in the algorithm.





REFERENCES

- [1] C. Bessiere, "Constraint propagation," in *Handbook of Constraint Programming*, Elsevier, 2006, pp. 29–83, doi: 10.1016/S1574-6526(06)80007-6.
- [2] F. Rossi, P. van Beek, and T. Walsh, "Chapter 4 constraint programming," in *Handbook of Knowledge Representation*, Elsevier, 2008, pp. 181–211, doi: 10.1016/S1574-6526(07)03004-0.
- [3] K. Schimmelpfeng and S. Helber, "Application of a real-world university-course timetabling model solved by integer programming," *OR Spectrum*, vol. 29, no. 4, pp. 783–803, Jul. 2007, doi: 10.1007/s00291-006-0074-z.
- [4] M. C. Chen, S. N. Sze, S. L. Goh, N. R. Sabar, and G. Kendall, "A survey of university course timetabling problem: perspectives, trends and opportunities," *IEEE Access*, vol. 9, pp. 106515–106529, 2021, doi: 10.1109/ACCESS.2021.3100613.
- [5] N. L. A. Aziz and N. A. H. Aizam, "A brief review on the features of university course timetabling problem," in *AIP Conference Proceedings*, 2018, vol. 2016, p. 020001, doi: 10.1063/1.5055403.





- [6] A. A. Lazarev and F. Werner, "A graphical realization of the dynamic programming method for solving NP-hard combinatorial problems," *Computers & Mathematics with Applications*, vol. 58, no. 4, pp. 619–631, Aug. 2009, doi: 10.1016/j.camwa.2009.06.008.
- [7] W. Chinnasri, S. Krootjohn, and N. Sureerattan, "Performance comparison of genetic algorithm's crossover operators on university course timetabling problem," in *Proceedings - 2012 8th International Conference on Computing Technology and Information Management*, ICCM 2012, 2012, vol. 2, pp. 781–786.
- [8] H. Babaei, J. Karimpour, and A. Hadidi, "A survey of approaches for university course timetabling problem," *Computers & Industrial Engineering*, vol. 86, pp. 43–59, Aug. 2015, doi: 10.1016/j.cie.2014.11.010.
- [9] P. Ramdé, P. Lapointe, and M. Dembélé, "The appropriation of policy changes in higher education by students in Sub-Saharan Africa: the case of the bachelor-master-doctorate Reform in Burkina Faso," *Comparative and International Education*, vol. 48, no. 2, pp. 1–19, Jun. 2020, doi: 10.5206/cie-eci.v48i2.10789.
- [10] E. K. Burke and G. Kendall, Eds., *Search methodologies*. Boston, MA: Springer US, 2014, doi: 10.1007/978-1-4614-6940-7.
- [11] D. Thierens, "Scalability problems of simple genetic algorithms" *Evolutionary computation* vol. 7, no. 4, pp. 331–352, Jun. 1999, doi: 10.1162/evco.1999.7.4.331.
- [12] P. Kostuch, "The university course timetabling problem with a three-phase approach," in *Practice and Theory of Automated Timetabling V. PATAT 2004*, Berlin: Springer, 2005, pp. 109–125, doi: 10.1007/11593577_7.
- [13] P. Laborie, "Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results," *Artificial Intelligence*, vol. 143, no. 2, pp. 151–188, Feb. 2003, doi: 10.1016/S0004-3702(02)00362-4.
- [14] D. Pisinger and M. Sigurd, "Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem," *INFORMS Journal on Computing*, vol. 19, no. 1, pp. 36–51, Feb. 2007, doi: 10.1287/ijoc.1060.0181.
- [15] M. Kumar, M. Husain, N. Upreti, and D. Gupta, "Genetic algorithm: review and application," *SSRN Electronic Journal*, 2010, doi: 10.2139/ssrn.3529843.
- [16] L. Davis, *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold, 1991.
- [17] E. K. Burke, D. G. Elliman, and R. F. Weare, "The automation of the timetabling process in higher education," *Journal of Educational Technology Systems*, vol. 23, no. 4, pp. 353–362, Jun. 1995, doi: 10.2190/NGYR-EXLB-RK79-K6NU.
- [18] G. Anandalingam, "Simulated annealing," in *Encyclopedia of Operations Research and Management Science*, New York, NY: Springer US, 2001, pp. 748–751, doi: 10.1007/1-4020-0611-X_956.
- [19] S. Abdullah, H. Turabieh, B. McCollum, and P. McMullan, "A hybrid metaheuristic approach to the university course timetabling problem," *Journal of Heuristics*, vol. 18, no. 1, pp. 1–23, Feb. 2012, doi: 10.1007/s10732-010-9154-y.
- [20] D. Abramson, "Constructing school timetables using simulated annealing: sequential and parallel algorithms," *Management Science*, vol. 37, no. 1, pp. 98–113, 1991.
- [21] E.-G. Talbi and T. Muntean, "Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem," in *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, vol. ii, pp. 565–573, doi: 10.1109/HICSS.1993.284069.
- [22] A. L. Bolaji, A. T. Khader, M. A. Al-Betar, and M. A. Awadallah, "University course timetabling using hybridized artificial bee colony with hill climbing optimizer," *Journal of Computational Science*, vol. 5, no. 5, pp. 809–818, Sep. 2014, doi: 10.1016/j.jocs.2014.04.002.
- [23] Y. Elhaddad and O. Sallabi, "A new hybrid genetic and simulated annealing algorithm to solve the traveling salesman problem," in *Proceedings of the World Congress on Engineering 2010*, 2010, pp. 11–14.
- [24] A. A. Mahiba and C. A. D. Durai, "Genetic algorithm with search bank strategies for university course timetabling problem," *Procedia Engineering*, vol. 38, pp. 253–263, 2012, doi: 10.1016/j.proeng.2012.06.033.
- [25] K. Wiliams and M. Ajinaja, "Automatic timetable generation using genetic algorithm," *Computer Engineering and Intelligent Systems*, vol. 10, no. 4, pp. 23–26, 2019.

BIOGRAPHIE OF AUTHORS



Maurice Comlan     received his Ph.D. degree from University of Nantes (France) in 2016. He has also master degree in 2012 from Polytechnic School of Abomey-Calavi in University of Abomey-Calavi (Bénin). He is currently a computer science researcher. His main researcher interest focus on real time system, formal method, optimization algorithms, and artificial intelligence. He can be contacted at maurice.comlan@uac.bj.



Corentin Allohoumbo     received his master degree in 2021 from Polytechnic School of Abomey-Calavi in University of Abomey-Calavi (Bénin). He is currently a computer science researcher. He works as a backend software engineer at Djamo, which is a Financial Software company with an estimated 110 employees. He can be contacted at corentinalcoy@gmail.com.